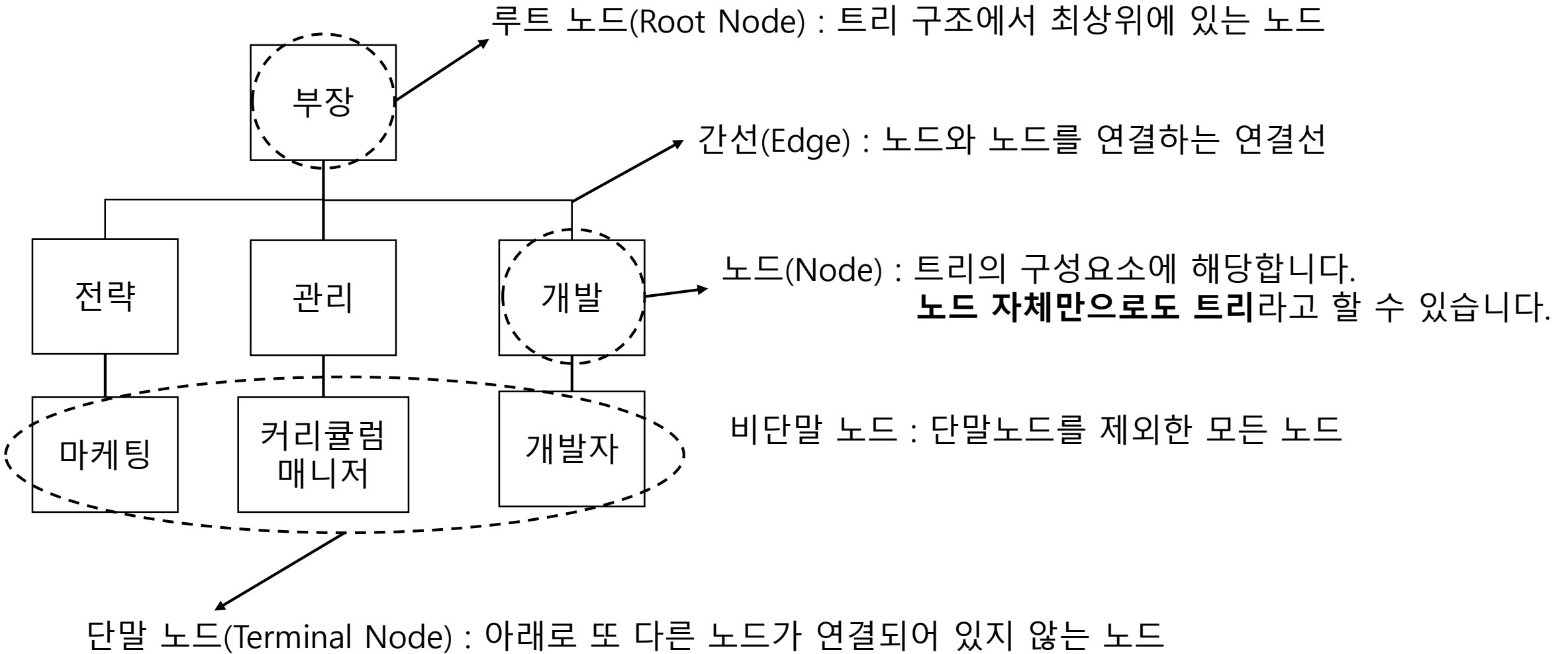


# 자료구조

1108 서민준

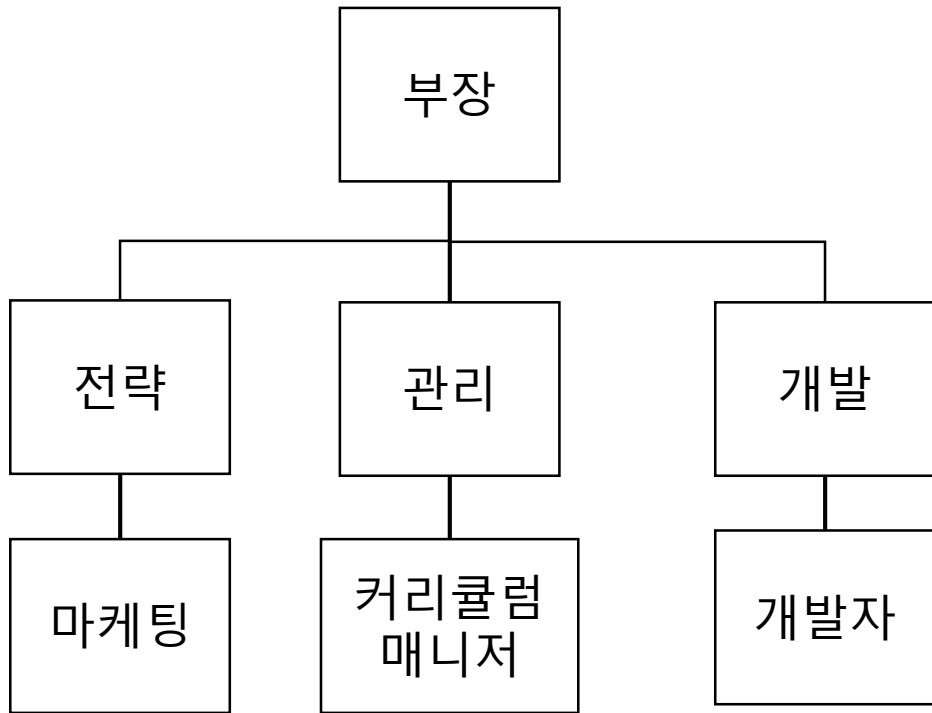
# 트리 (1)



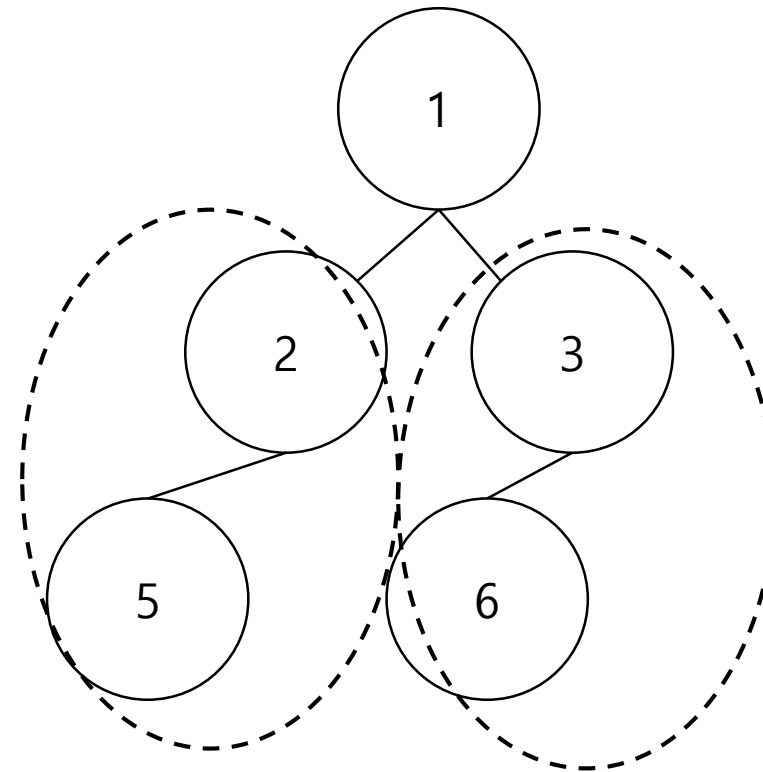
# 트리 (2)

차수(Degree) : 자식 노드의 수

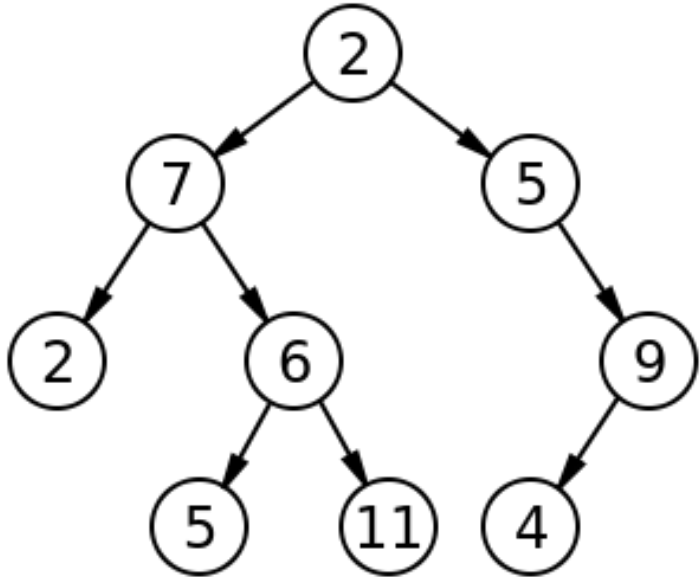
높이 : 노드의 차수 중 가장 큰 차수



서브 트리(Sub Tree)



# 이진 트리 (1)



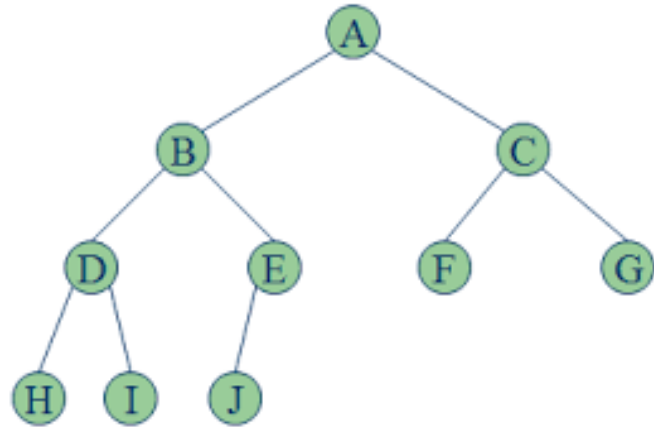
정의

자식 노드를 최대 2개까지 가질 수 있고, 자식 노드는 공집합 (NULL)일 수도 있는 트리

조건

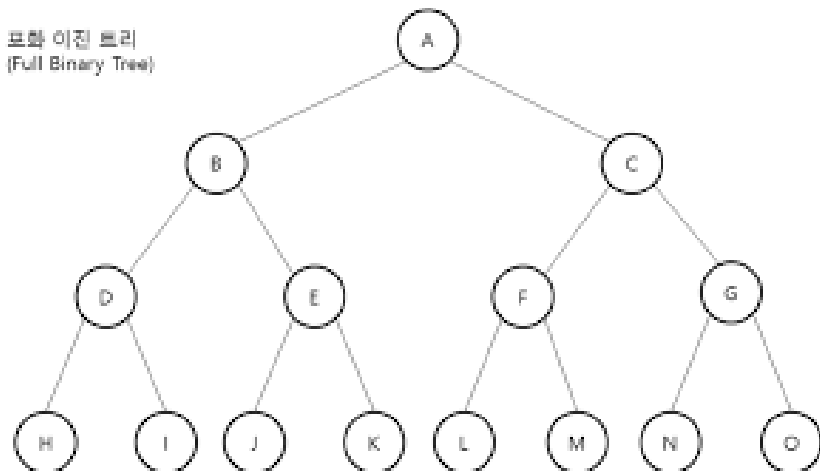
1. 루트 노드를 중심으로 두 개의 서브 트리로 나뉘어진다.
2. 나뉘어진 두 서브 트리도 모두 이진 트리이어야 한다.

# 이진 트리 (2)



완전 이진 트리

위에서 아래로, 왼쪽에서 오른쪽으로 빈틈없이 노드가 채워져 있는 상태의 이진 트리

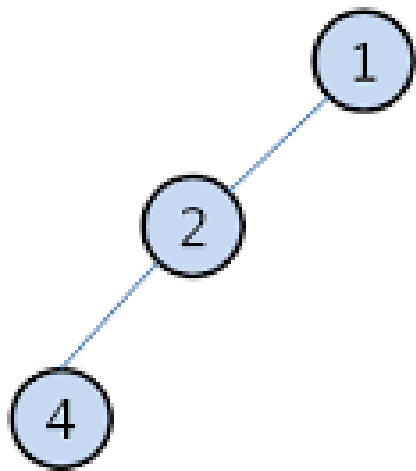


포화 이진 트리

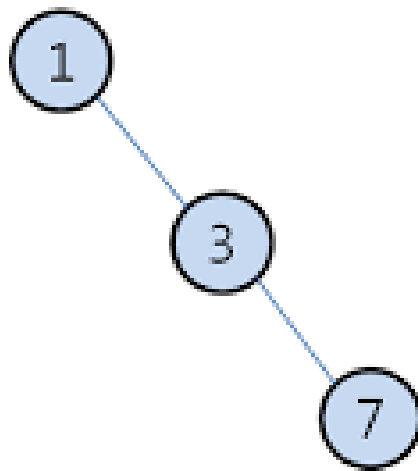
완전 이진 트리 상태 중 하나로, 모든 레벨에 노드가 다 채워져 있는 트리

레벨의 최대 노드 수 :  $2^{level}$

# 이진 트리 (3)



왼쪽편향 이진트리



오른쪽편향 이진트리

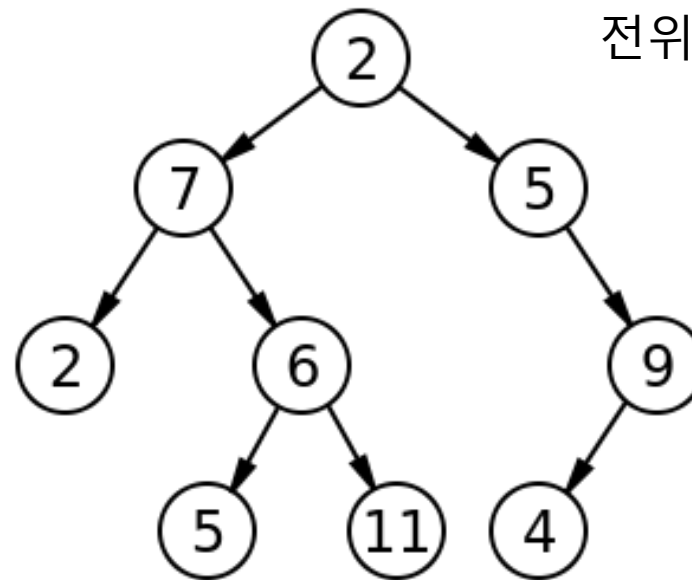
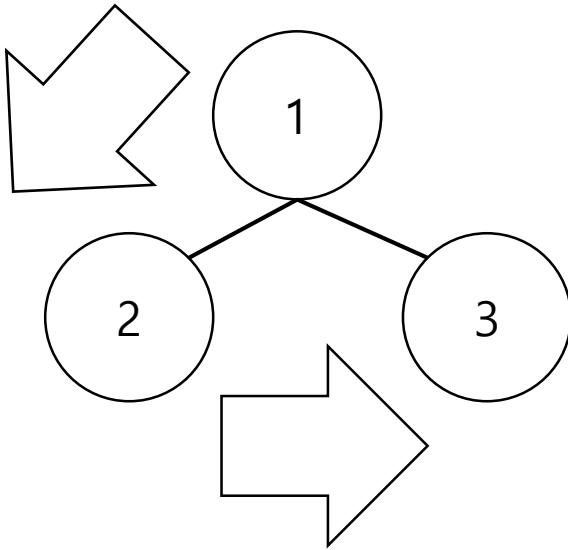
편향 이진 트리

노드가 **왼쪽** 혹은 **오른쪽**으로만 연결되어 있는 이진 트리

# 이진 트리 순회 (1)

순회 : 트리에 존재하는 모든 노드를 도는 방법

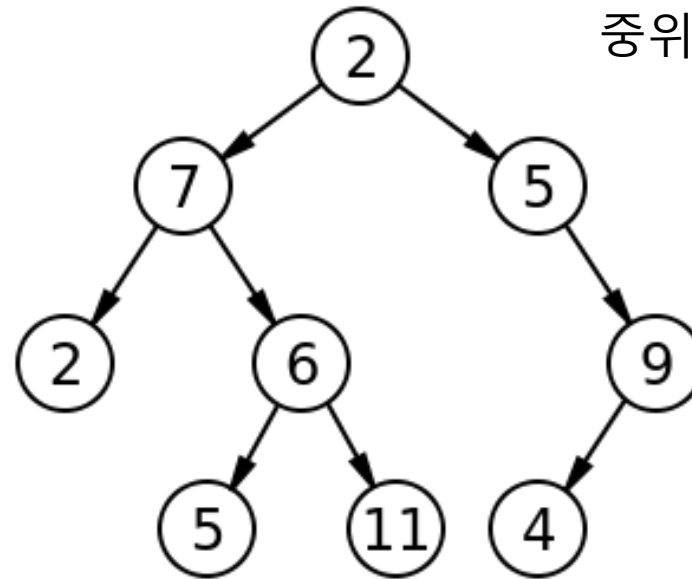
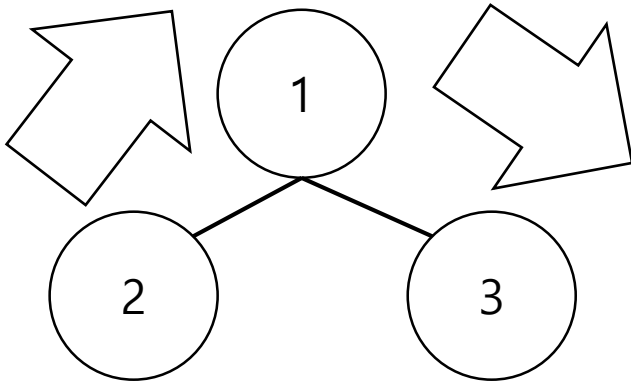
1. 전위 순회 : 루트 노드를 가장 먼저 돈다.



전위 순회 결과는?

# 이진 트리 순회 (2)

2. 중위 순회 : 루트 노드를 2번째로 돈다.

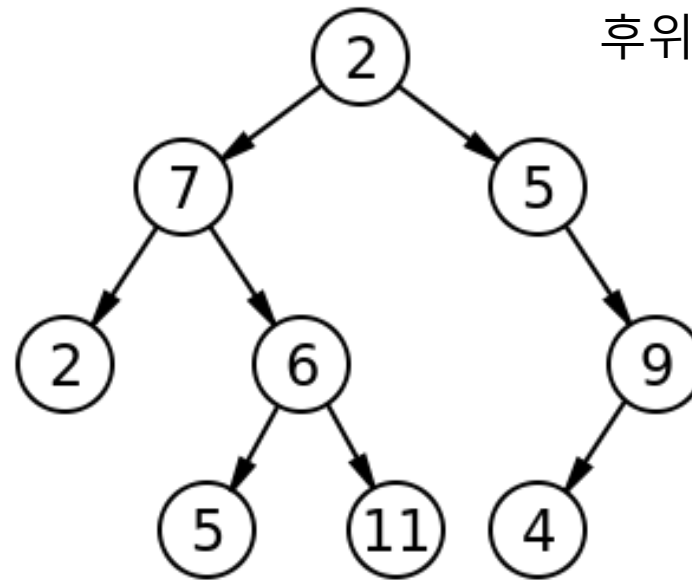
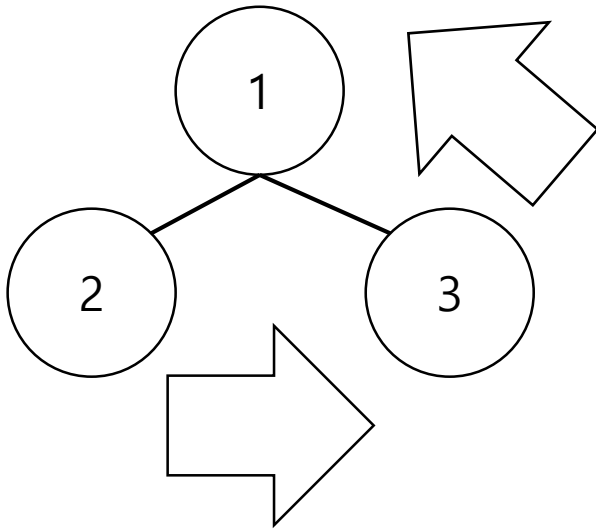


중위 순회 결과는?



# 이진 트리 순회 (3)

3. 후위 순회 : 루트 노드를 가장 마지막에 돈다.

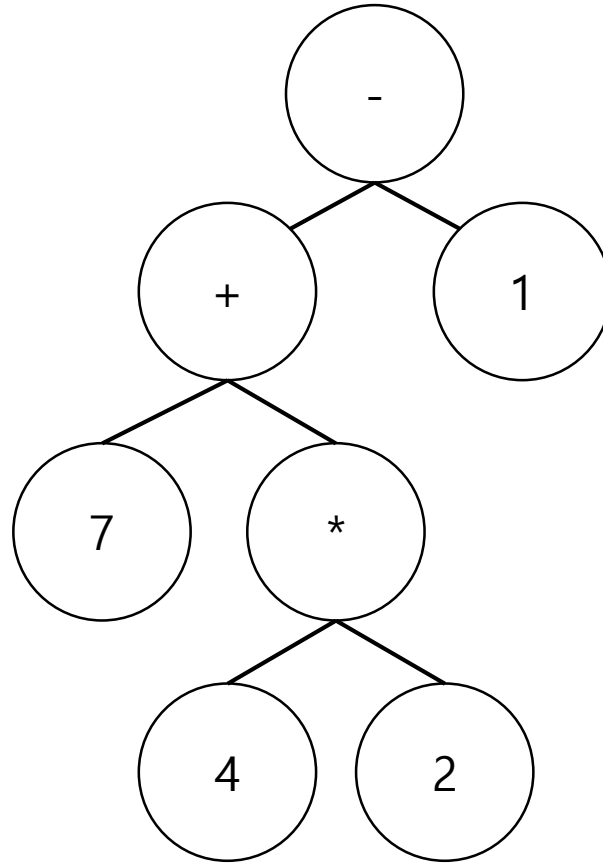


후위 순회 결과는?

# 수식 트리 (1)

중위 표현식  
 $7 + 4 * 2 - 1$

트리화

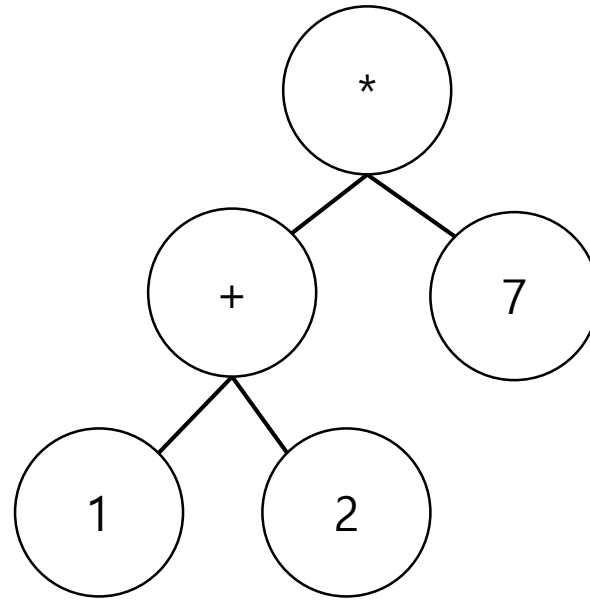
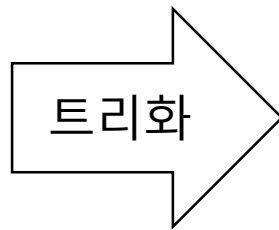


대체 어떤 규칙으로 생성되었는가?

# 수식 트리 (2)

중위 표현식으로 표현된 것을 바로 트리로 만드는 것보다  
후위 표현식으로 표현된 식을 트리로 표현하는 것이 더 쉽다.

후위 표현식  
1 2 + 7 \*

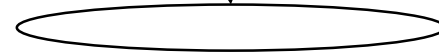
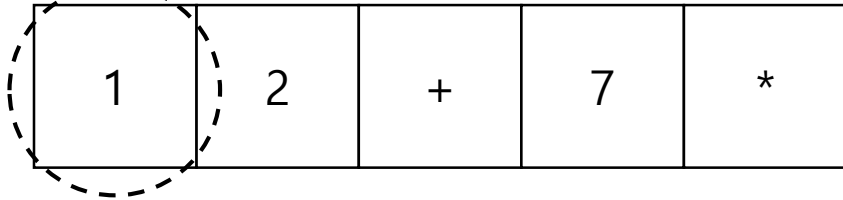


뭔가 규칙이 보이지 않는가?

# 수식 트리 (3)

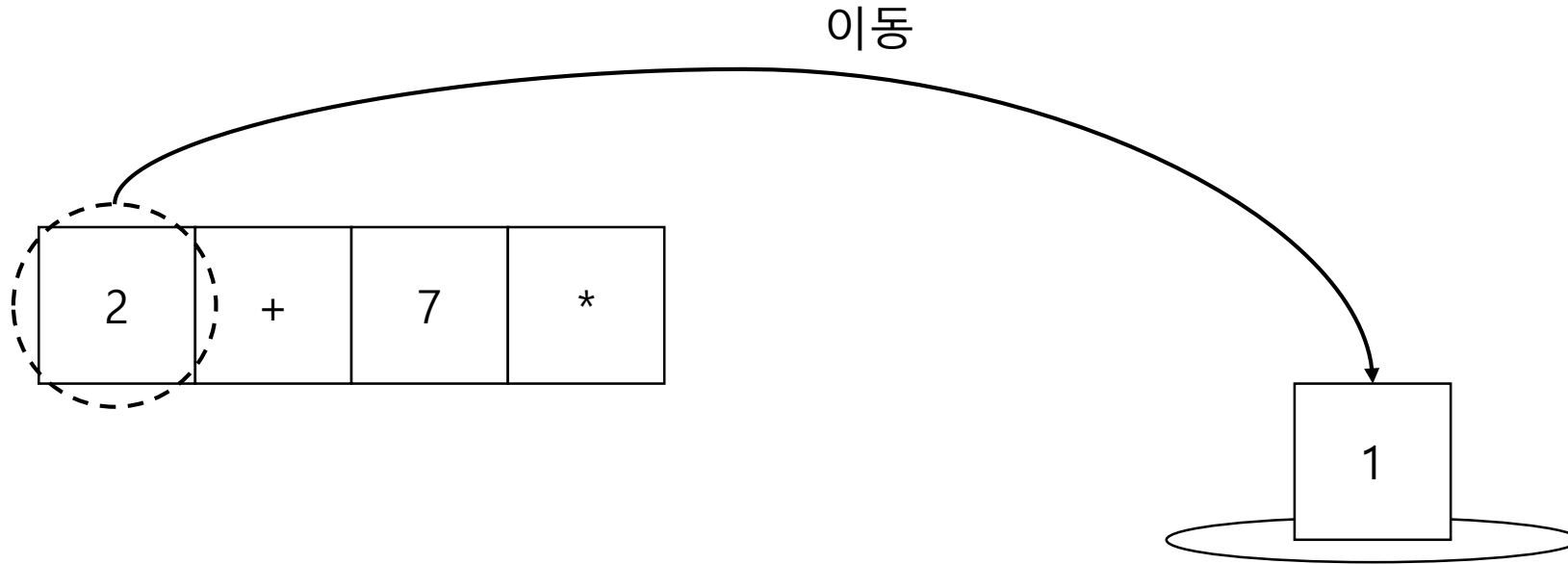
후위 표현식  
1 2 + 7 \*

이동



나는 접시다.

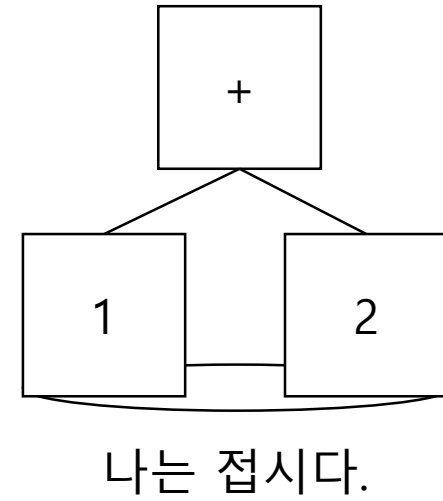
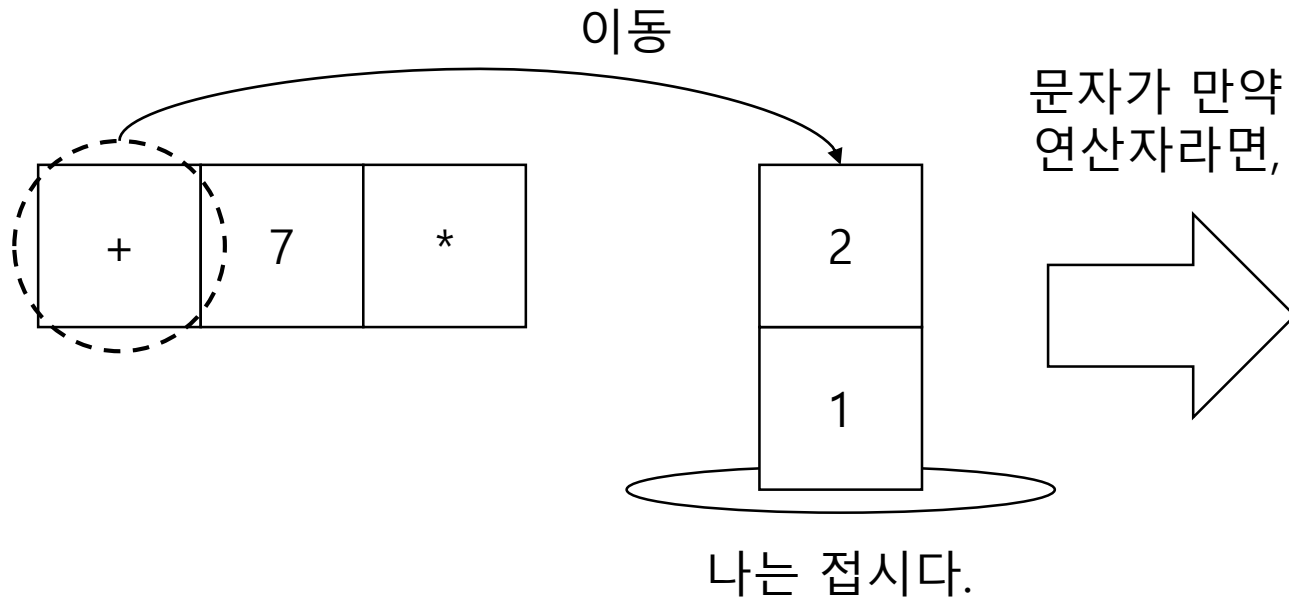
# 수식 트리 (4)



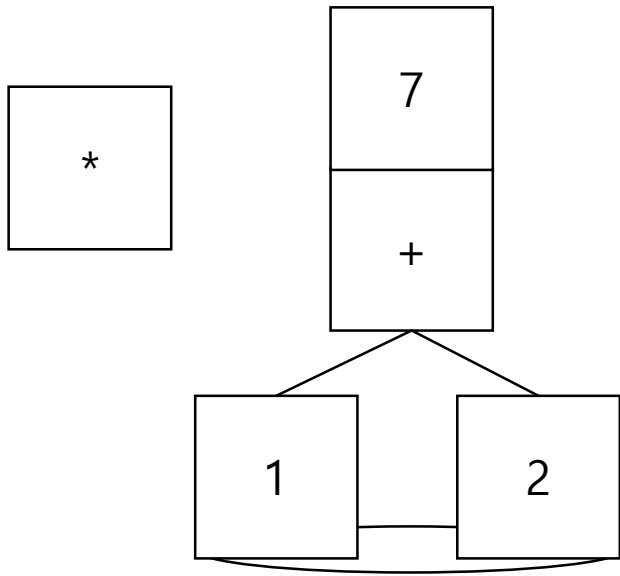
피연산자는 무조건 접시(스택)으로 옮긴다.

나는 접시다.

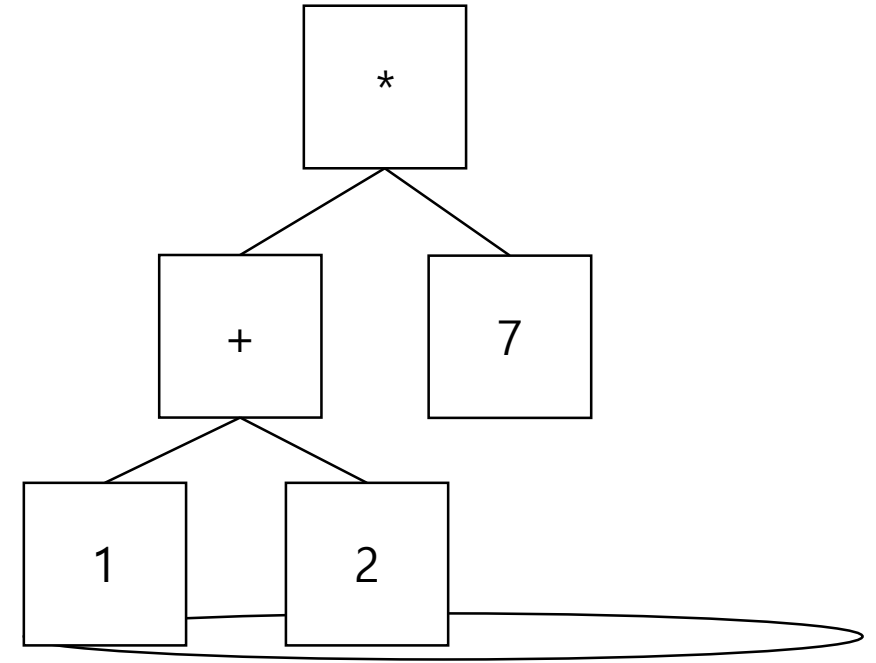
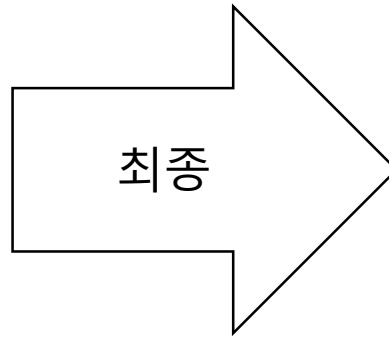
# 수식 트리 (5)



# 수식 트리 (6)



나는 접시다.



나는 접시다.

# 수식 트리 (7)

수식 트리를 만드는 과정 정리!

- 피연산자를 만나면 무조건 접시(스택)으로 옮긴다.
- 연산자를 만나면 스택에서 두 개의 피연산자를 꺼내어 자식 노드로 연결
- 자식 노드를 연결해서 만들어진 트리는 다시 접시(스택)으로 옮긴다.

순회 방법에 따라 표기되는 식이 달라짐

전위	순회	시	전위	표기법
중위	순회	시	중위	표기법
후위	순회	시	후위	표기법

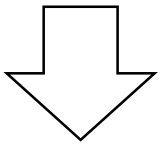


# 수식 트리 (8)

중위 식으로 표기할 때 소괄호를 출력하는 방법

후위 표현식

1 2 + 7 \*



중위 표현식

(( 1 + 2 ) \* 7 )

→ 연산자의 수와 소괄호의 수가 일치한다!

```
void ShowInfixTypeExp(BTreeNode* bt)
{
    if (bt == NULL)
        return;

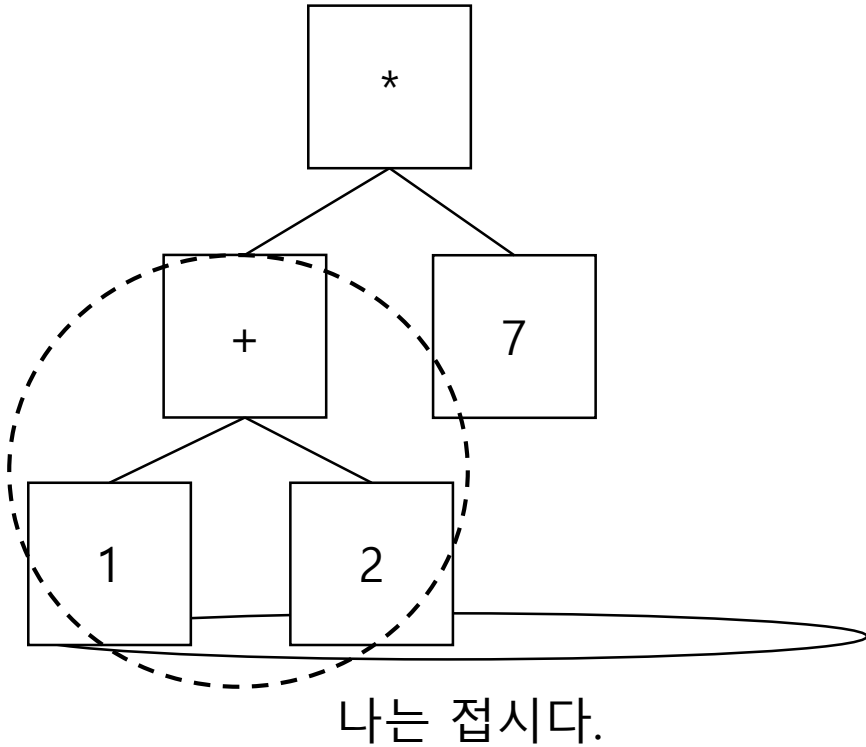
    if (bt->left != NULL || bt->right != NULL)
        fputs("(", stdout);

    ShowInfixTypeExp(bt->left);
    printf("%c ", bt->data);
    ShowInfixTypeExp(bt->right);

    if (bt->left != NULL || bt->right != NULL)
        fputs(")", stdout);
}
```

# 수식 트리 (9)

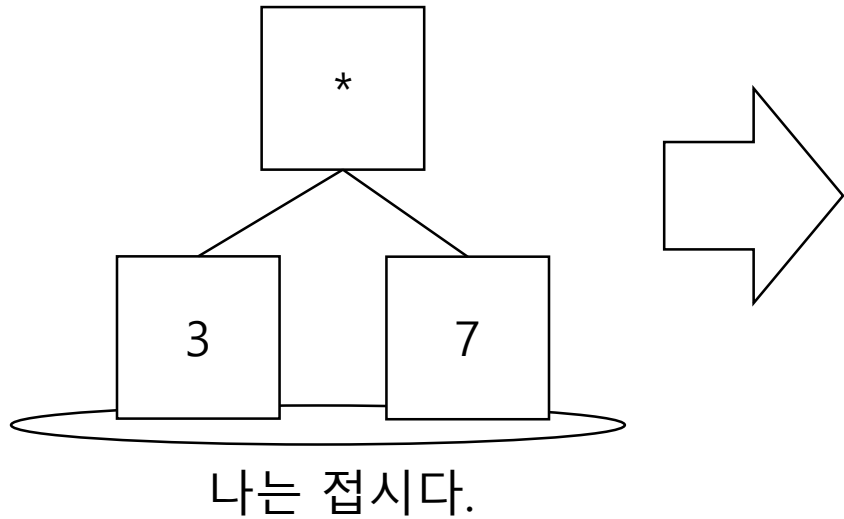
그럼 이제 만들어진 트리를 계산해보자.



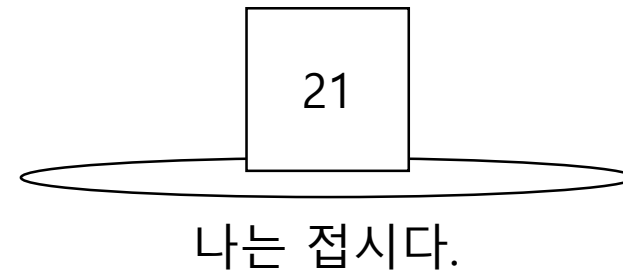
1. 가장 아래에 있는 1과 2를 + 연산을 하고 그 값을 반환한다.  
후위 순회로 돌면 1 2 +가 된다.

# 수식 트리 (10)

2. 다시 후위 순회를 하여 연산을 진행한다.



3. 남은 것이 루트 노드 하나이므로 루트 노드를 출력한다.



# 수식 트리 (11)

```
int EvaluateExpTree(BTreeNode* bt)
{
    int op1, op2;

    if (bt->left == NULL && bt->right == NULL) // 단말 노드라면
        return bt->data;

    // 피연산자를 구한다.
    op1 = EvaluateExpTree(bt->left);
    op2 = EvaluateExpTree(bt->right);

    switch (bt->data) // 연산자의 종류별로 연산 진행 후 반환
    {
        case '+':
            return op1 + op2;
        case '-':
            return op1 - op2;
        case '*':
            return op1 * op2;
        case '/':
            return op1 / op2;
    }
    return 0;
}
```

# 우선순위 큐 (1)

일반적인 큐와 동일한 연산을 가진다.

enqueue : 우선순위 큐에 데이터를 삽입하는 행위

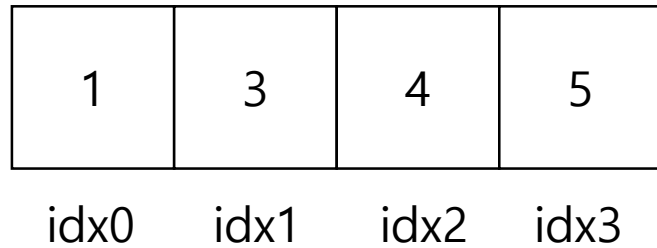
dequeue : 우선순위 큐에서 데이터를 꺼내는 행위

단, 연산의 결과에서 차이가 있다.

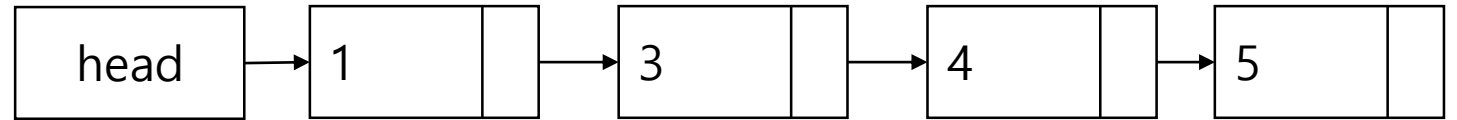
우선순위 큐는 **들어간 순서에 상관없이, 우선순위가 높은 데이터가 먼저 나온다.**

# 우선순위 큐 (2)

배열로 구현하는 우선순위 큐



연결 리스트 기반의 우선순위 큐

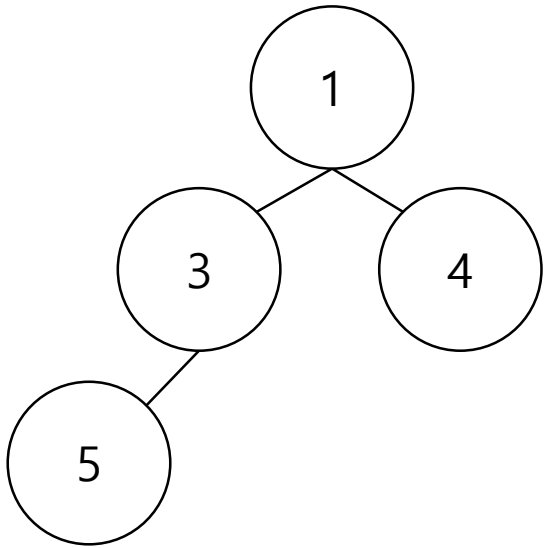


단점

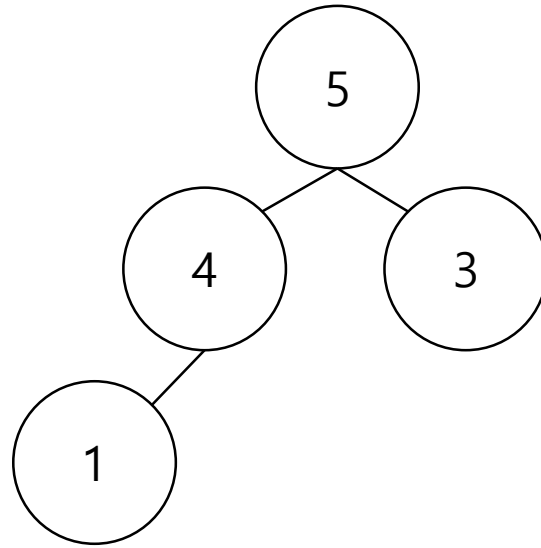
데이터를 삽입 / 삭제 과정에서 데이터를 shift하는 연산을 수반해야 한다.  
삽입의 위치를 찾기 위해서 배열에 저장된 모든 데이터와 우선순위 비교를 해야 할 수도 있다.

# 우선순위 큐 (3)

힙으로 구현하는 우선순위 큐



최소 힙



최소 힙

# 힙 (1)

완전 이진 트리의 형태

모든 노드가 **우선순위에 따라 정렬**되어 있는 상태의 **이진 트리**

부모 노드가 자식 노드보다 **작거나 같으면 최소 힙**

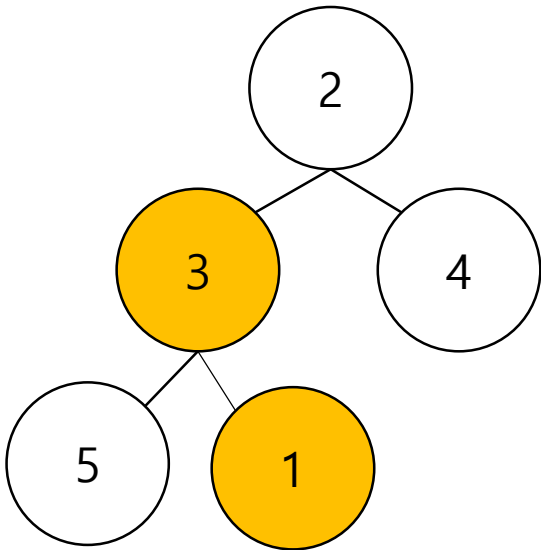
부모 노드가 자식 노드보다 **크거나 같으면 최대 힙**



# 힙 (2)

여기서는 최소 힙(min heap)을 기준으로 설명합니다.

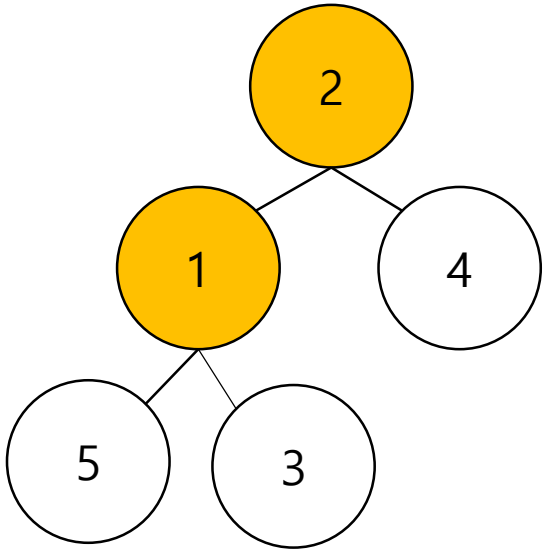
새로운 데이터는 우선순위가 제일 낮다는 가정하에 '마지막 위치'에 저장한다.



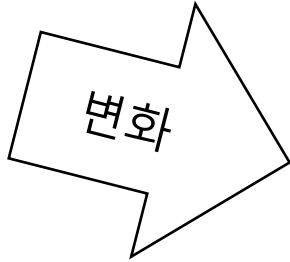
데이터를 삽입하였다면 자신의 부모 노드와 우선순위를 비교한다.  
-> 새로 삽입된 자료가 우선순위가 더 높다.

만약 새로 삽입된 자료의 우선순위가 높다면 부모 노드의 값과 자식 노드의 값을 변경한다.

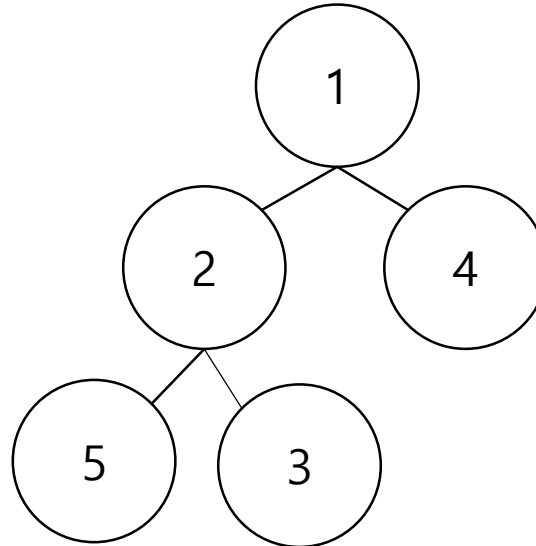
# 힙 (3)



아직 부모 노드가 존재하므로 부모 노드와 다시 한 번 우선순위를 비교한다.  
-> 이번에도 우선순위가 자식 노드가 더 높다.

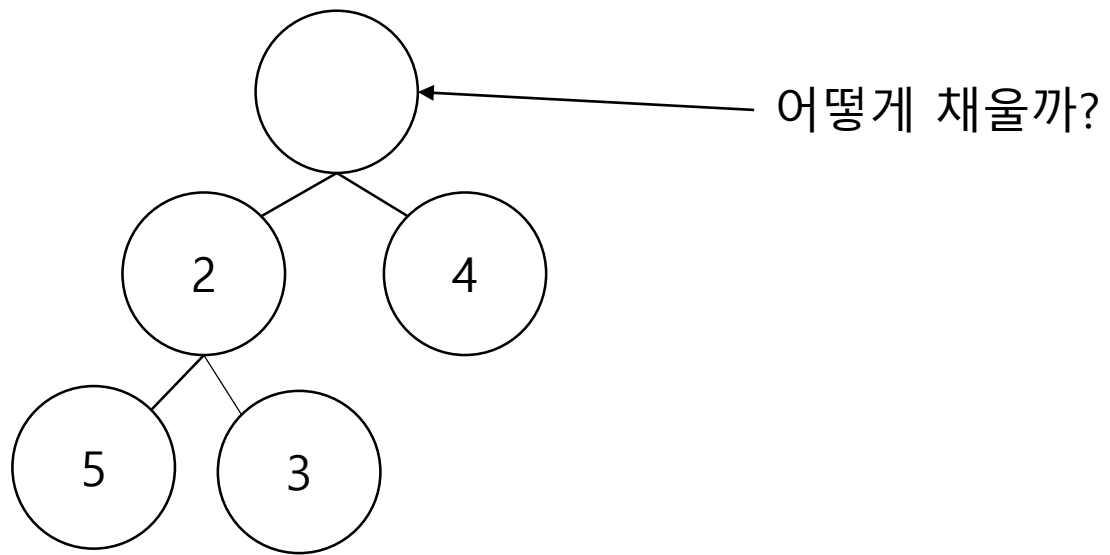


최종결과

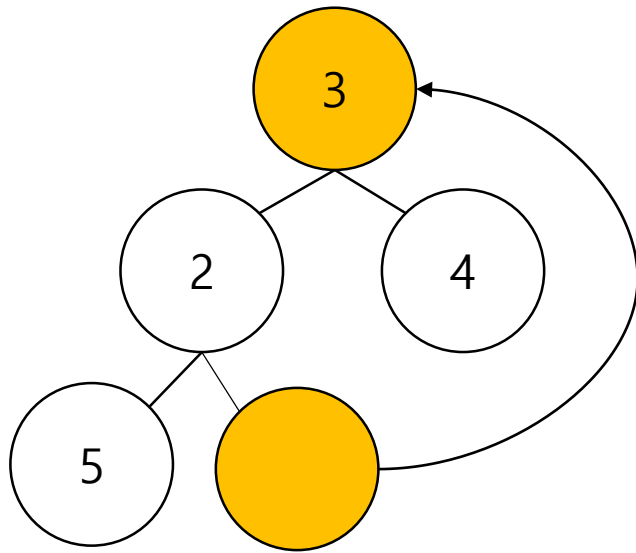


# 힙 (4)

이번에는 데이터를 삭제하는 과정을 알아보자.

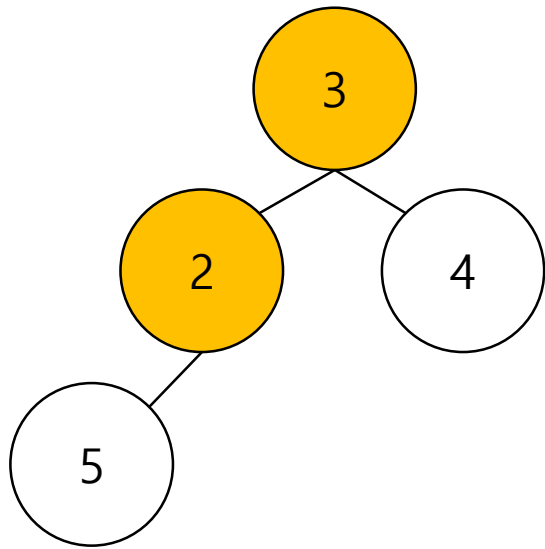


# 힙 (5)



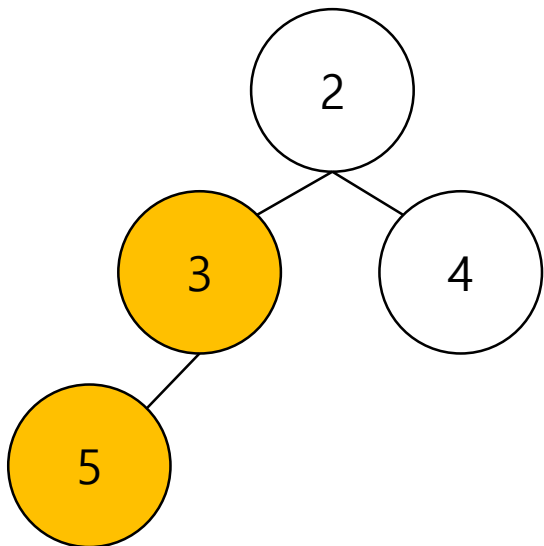
우선 가장 마지막에 있는 노드를 루트 노드로 옮긴다.

# 힙 (6)



그 뒤, 자식 노드와 우선순위를 비교하여 자신의 위치를 찾는다.  
-> 현재 자식의 우선순위가 더 높으므로 위치 변경이 필요하다.

# 힙 (7)



위치 변경을 한 후에도 자식 노드가 존재하므로 다시 한 번 우선순위를 검사한다.  
-> 이번에는 자식 노드가 우선순위가 더 낮으므로 자리를 찾음